HPC101 实验报告 2

实验二：向量化计算

姓 名: 洪 奕 迅

学 号: 3230102930

班 号: 工信 2319

(短学期, 2024)

浙江大学

计算机学院

2024 年 5 月 22 日

目录

1 双线性插值的向量化实现

Theorem 1.0.1 ▶ 向量化计算

向量化计算是对于传统计算的一种优化方式，其中最核心的特点在于数据操作的对象不再是一个单独的标量值而是向量和矩阵（或者按照数据结构的说法——数组）。因此比起显式的循环迭代，向量化以一种并行的方式大大加速了数据操作。而在本次实验中向量化计算基于 Python 的 numpy 计算库。

1.1 双线性插值

线性插值的思路就是假设一个离散集合的相邻点间符合线性，从而我们能够通过线性插值的方式计算其中某点的值，而双线性插值则是将这一思路扩展到了二维平面的每一个小方格上。

基于 */starter_code/bilinear_interp/baseline.py*，我们可以知道本实验的插值取得是相邻四个整数点，而基于这个假设和此前从文档中摘录的数学表达式，我们对本题的插值表达式进行一些推导：

我们想得到未知函数 f 在点 $P=(x,y)$ 的值，记 $\overline{x}$ 为 x 的小数部分，于是有：

$$Q_{11}=([x],[y]),\ Q_{12}=([x],[y]+1),\ Q_{21}=([x]+1,[y]),\ Q_{22}=([x]+1,[y]+1)$$

$$f(x,y)=\begin{bmatrix}1-\overline{x} & \overline{x}\end{bmatrix}\begin{bmatrix}f(Q_{11}) & f(Q_{12})\\ f(Q_{21}) & f(Q_{22})\end{bmatrix}\begin{bmatrix}1-\overline{y}\\ \overline{y}\end{bmatrix}$$

而为了进行向量化操作，所有的被操作数都要被扩展成对应的数组，并且我们需要扩展部分数组维数来确保其能满足 numpy 的广播原则，即在前几个维度相同时，要进行计算，则需要扩展出最后一个维度，并且维数应当为 1 或与另一个操作向量相等。

不妨做以下向量组定义：

$$Q_{xx}=\{f_i(Q_{xx})\},\ \overline{X/Y}=\{\overline{x_i/y_i}\}$$

则当其带入此前的表达式以后对应的则对应数学中的分块矩阵计算，我们将外层的矩阵展开后就得到了一个标准向量化计算的表达式。

1.2 代码实现

在开始进行具体的代码设计之前，我们还需要分析此前提到的扩展维度问题，首先是对于传入矩阵中的单元 $a[N, W, H]$，在没有向量化之前这个值代表矩阵中某个点的各通道值的集合，而向量化后我们希望能将其变为所有点的值构成的矩阵。这对于后两者是极其自然的，我们将直接从目标坐标矩阵 b 中取所有单元的 x, y 坐标，这是符合其在几何上的意义的。而 N 则不同，其代表需测试的矩阵的数量，这显然不可能有同横纵坐标一样的任何三维（实际上是多组与二维坐标的组合）含义，因此我们需要对齐进行维度的扩展。

参与计算的是传入矩阵的所有单元的所有通道的值，因此在上一段讨论的单元其实还有第四维 C，代表各个通道的值，而我们从 b 中取出的坐标集只是单纯的所有点的坐标，并没有第四个维度，而由于所有通道都参与同样形式的计算，我们只需要把其进行 $x[...:none]$ 形式的扩展即可。

至此我们完成了所有计算层面和代码层面的问题，以下是我设计的向量化的双插值代码：

Code Snippet 1.2.1 ▶ starter_code/bilinear_interp/vectorized.py

```
import numpy as np
from numpy import int64
def bilinear_interp_vectorized(a: np.ndarray, b: np.ndarray) -> np.ndarray:
    # get axis size from ndarray shape
    N, H1, W1, C = a.shape
    N1, H2, W2, _ = b.shape
    assert N == N1
    # TODO: Implement vectorized bilinear interpolation
    res = np.empty((N, H2, W2, C), dtype=int64)
    # X and y values of all the target nodes from the matrix b
    x = b[...,0]
    y = b[...,1]
    #Integer part of x and y
    x_idx = np.floor(x).astype(int64)
    y_idx = np.floor(y).astype(int64)
    #Decimal part of x and y
    _x = x-x_idx
    _y = y-y_idx
    #Extenion of the n and _x, _y dimension to match x and y
    n_idx=np.arange(N)[:, None, None]
    _x = _x[...,None]
    _y = _y[...,None]
    #Get all the nearby points
```

```
    q11 = a[n_idx, x_idx, y_idx]
    q21 = a[n_idx, x_idx + 1, y_idx]
    q12 = a[n_idx, x_idx, y_idx + 1]
    q22 = a[n_idx, x_idx + 1, y_idx + 1]
    # Perform actual bilinear interpolation
    res = (q11 * (1 - _x) * (1 - _y) +
           q21 * _x * (1 - _y) +
           q12 * (1 - _x) * _y +
           q22 * _x * _y).astype(int64)
    return res
```

2 正确性验证和性能测试

这里使用提供的 *main.py* 进行了简单的正确性验证和性能测试：

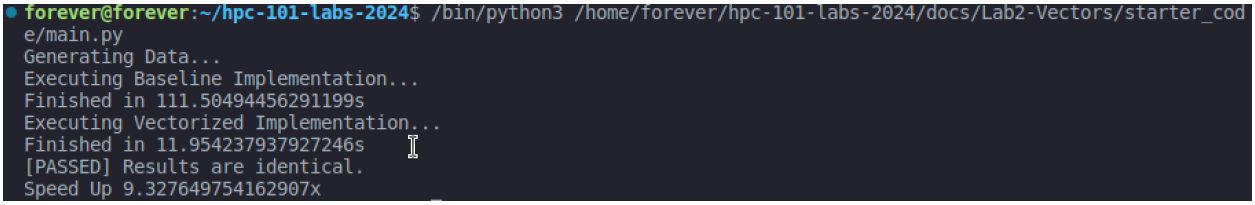

图 2.1: 正确性验证和性能测试

后续进行了多次测试，在本机（Macbook Air-M2，Ubuntu18.04 - 4core 4G）下的平均加速倍率:

$$\overline{Speed} = \frac{Vectorized}{Baseline} = 9.723X$$

同时注意到这个加速倍率同本机性能有关，在本机（M2）和 7800X3D 平台上分别达到了平均约 30X 和 35X 的加速倍率。

A 代码

所有代码已被上传至 Github 本地克隆仓库：ForeverHYX